Text phishing detection project

Tuvia Smadar - 315638577
Shira yogev - 325877108
Itai Mizlish - 211821913

Introduction

Phishing attacks are a common threat in the digital world, where attackers try to trick people into sharing personal information. This project focuses on detecting phishing messages in emails, using deep learning techniques to classify whether the email is legitimate or a phishing attempt.

Data Sources

Phishing datasets compiled from various resources for classification and phishing detection tasks. Datasets correspond to a compilation of 2 sources, which are described below:

- **Emails:**

  Over 18,000 emails from Enron Corporation employees, used to detect phishing emails through machine learning.

- **SMS Messages:**

  A collection of 5,971 SMS messages, including spam, smishing, and ham messages.

The dataset has been originally preprocessed by the publisher to eliminate null, empty, and duplicate entries. Additionally, class balancing has been performed to ensure the model is not biased toward any specific class.

Dataset and Task Overview

Problem: This is a classification problem where the goal is to classify email and SMS texts as either phishing or legitimate.

Features:

The dataset contains one feature-the column of text, which represents the emails and SMS text messages.

Text data is transformed into numerical features using techniques such as Bag-of-Words, TF-IDF, and word embeddings. More advanced models like BERT and

FastText (that used in this project) automatically generate contextualized embeddings, capturing the meaning of words in their specific contexts.

After preprocessing that can be added or done automatically, the next step is tokenization, where the text is broken down into individual words or “tokens.” This step breaks down the text content into discrete units that can be analyzed and processed.

Label: The labels are binary, with 0 representing legitimate text and 1 representing phishing text.

Goal: The goal of the project is to explore deep learning techniques and develop a model capable of detecting phishing messages within texts.

Data Split: The dataset has been split into training and test sets, with 80% of the data allocated for training the model and the remaining 20% used for testing and evaluating its performance.

Evaluation Metrics: The evaluation metrics we use is:

$$Precision = \frac{True\ Positives(TP)}{True\ Positives(TP) + False\ Positives(FP)}$$

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}$$

$$F1 - Score = 2\,\frac{Precision * Recall}{Precision + Recall}$$

$$Recall = \frac{True\ Positives(TP)}{True\ Positives(TP) + False\ Negatives(FN)}$$

We use **accuracy**, **precision**, **recall**, and **F1-score** to evaluate models because each metric provides a unique perspective on performance. While accuracy gives an overall measure of correctness, precision and recall focus on the trade-offs between false positives and false negatives, which are critical in high-stakes applications. The F1-score balances precision and recall, making it ideal for assessing models in scenarios where both types of errors carry significant consequences for example in our situation Email can be blocked due false negative.

Baseline:

Dummy Classifier:

DummyClassifier is a baseline for comparison with more complex models. In this project, it is initialized with the "most_frequent" strategy. That means the classifier will always predict the most frequent class present in the training data. Therefore, if most of the training data contains non-phishing messages, it will always predict non-phishing for all cases, regardless of the input.

This model will train the training data, X_train and y_train, but the model actually doesn't learn any pattern in this dummyClassifier; it just memorizes the majority class.

After training, the model predicts on the test set, which is X_test. Since it's a classifier that will predict the majority class for all test instances, all of the predictions will turn out to be the same.

Classification Report:

	precision	recall	f1-score	support
0	0.61	1.00	0.76	2469
1	0.00	0.00	0.00	1559
accuracy			0.61	4028
macro avg	0.31	0.50	0.38	4028
weighted avg	0.38	0.61	0.47	4028

Accuracy: 0.6130

The Dummy Classifier predicts the common category (non-phishing) well, but fails completely to detect phishing overall, the recall is 100% because there is no false negatives, only false positives and true positives.

Logistic Regression for Text Classification

In this project, we will use Logistic Regression to classify text data (phishing vs non-phishing). The text data is first converted into numeric features using methods like Bag of Words and TF-IDF. Logistic regression is then trained to predict whether a given text is phishing or not.

TF-IDF

is a simple model for converting text into numeric features. The models learns the importance of words compared to all documents

TF-IDF made up of two metrics:

- Term Frequency (TF): measures the frequency at which a word appears in a document and helps in determining the important words in a document.
- Inverse Document Frequency (IDF): A measure of the importance of a word—is given a higher weight for rare words and a lower weight for common words.

$$TF = \frac{Number\ of\ times\ a\ word\ "X"\ appears\ in\ a\ Document}{Number\ of\ words\ present\ in\ a\ Document}$$

$$IDF = log\left(\frac{Number\ of\ Documents\ present\ in\ a\ Corpus}{Number\ of\ Documents\ where\ word\ "X"\ has\ appeared}\right)$$

$$TF\ IDF\ =\ TF * IDF$$

How to Implement a Model

Step 1: TF-IDF Vectorization

The TfidfVectorizer from the sklearn library is used to convert the raw text data into a TF-IDF matrix, where each word is represented numerically. This function use in max_features=150000 parameter that limits the number of features in order to make sure that the model only considers the most important words.
The output matrix represents how important each word is relative to the other words across all documents.

Step 2: Train-Test Split

The dataset is then split into two sets: one for training (80%) and the other for testing(20%). The `random_state=42` ensures that the split is reproducible.

Step 3: Convert Data to PyTorch Tensors

The TF-IDF matrix is converted to PyTorch tensors with `torch.tensor` so that it can be used with the PyTorch model for training.It specifies that the data should be of type float with `dtype=torch.float32`

Step 4: Defining the Logistic Regression Model

The LogisticRegressionModel class is defined using PyTorch's nn.Module. The model has a linear layer nn.Linear that performs the transformation of the input data into a single output used for binary classification between phishing and non-phishing. And a sigmoid function nn.Sigmoid that applied to the output of the linear layer in order to transform the raw output into probabilities between 0 and 1.

Step 5: Model Initialization and Loss Function

The model is initialized with `input_size`, which is determined by the number of features in the training data (`X_train_tensor.shape[1]`).

Also, the model defines the loss function as Binary Cross-Entropy Loss (`BCELoss`). It measures the difference between the predicted probabilities and the actual labels (0 or 1). The goal of training is to minimize this loss. Next, The Adam optimizer is defined with a learning rate of 0.001 and `model.parameters()` that refers to the parameters (weights) of the model. The Adam optimizer is an optimization algorithm that adapts the learning rate for each parameter based on the first and second moments of the gradients. This helps the model converge faster and avoid issues with high variance during training. The optimizer is responsible for updating the parameters to minimize the loss during training.

Step 6: Training the Model

The model is trained in 10 epochs with a random permutation of the training data.
For each batch:
The optimizer is zeroed `(optimizer.zero_grad())` to clear previous gradients. Then, performs a forward pass to compute the predictions. The model computes the loss, and then the model performs a backward pass `(loss.backward())` to compute gradients. Next, the optimizer updates the model's weights `(optimizer.step())` and the loss is printed at the end of each epoch to track the model's progress.

Step 7: Evaluating the Model

After training, the model is evaluated on the test set to compute accuracy and generate the classification report.

The model is set to evaluation mode with `model.eval()` and no gradients are computed with `torch.no_grad()`. The output probabilities converted to binary predictions based on a 0.5 threshold. The accuracy and the classification report is printed, showing the next table:

```
Accuracy: 0.9553

Classification Report:

              precision    recall  f1-score   support

  Legitimate       0.94      0.99      0.96      2493
    Phishing       0.98      0.90      0.94      1535

    accuracy                           0.96      4028
   macro avg       0.96      0.95      0.95      4028
weighted avg       0.96      0.96      0.95      4028
```

Comparison to baseline

The DummyClassifier has a low accuracy of 61.3% and fails to predict phishing messages, with zero true positives for phishing. On the other hand, the TF-IDF with Logistic Regression model has an accuracy of 95.53%, showing good performance in classifying messages. Precision, recall, and F1-score for both classes are higher than 94%, indicating a strong performance even for the minority class.

Legitimate messages (class `0`): The Logistic Regression model achieves a **precision of 0.94**, a **recall of 0.99**, and an **F1-score of 0.96**, indicating that it effectively identifies legitimate messages with high accuracy.

Phishing messages (class `1`): The model achieves a **precision of 0.98**, a **recall of 0.90**, and an **F1-score of 0.94**, showcasing its ability to detect phishing messages accurately, even as the minority class.

Results underline the high accuracy of the TF-IDF model, which is able to distinguish phishing messages from legitimate ones, in contrast to DummyClassifier, which basically predicts the majority class.

Neural Network:

Simple Classifier

The Simple Classifier is a simple Neural Network.The text data is converted into numeric features, and a PyTorch based model is trained to make predictions.

How the Model Works

Step 1: Model Definition

A simple feedforward neural network (`SimpleClassifier`) is defined using PyTorch. It consists of an input Layer that takes numeric features as input, a Fully Connected Layer that maps the input features to the number of output classes, and an output that produces raw logits (unnormalized scores). These logits are used by the loss function (`nn.CrossEntropyLoss`), which internally applies softmax to calculate probabilities.

Step 2: Data Preparation

The dataset is split into train and test sets. The numeric features are converted into PyTorch tensors, and the labels are converted into numeric form using TF-IDF, compatible with PyTorch's loss functions.

Step 3: Model Initialization

In this model, the Input Dimension is set to the number of features in the training dataset, represented as `input_dim = X_train.shape[1]`. The Output Classes is just the number of unique labels from the target variable, represented as `num_classes = len(torch.unique(y))`. For the Loss Function, use Cross-Entropy Loss, which is achieved with `criterion = nn.CrossEntropyLoss()`—a measure that calculates the difference between the predicted labels and the actual labels.

The Optimizer used is the Adam optimizer, initialized with a learning rate of 0.001 through `optimizer = optim.Adam(classifier.parameters(), lr=0.001)`, to update the gradients efficiently during training.

Step 4: Training the Model

At the beginning, It shuffles the data of every epoch, using `torch.randperm(X_train.size(0))`, which generates a random permutation of the indices. This will make sure that the model will not see the data in the same order each time, and hence it generalizes better. Then, it processes the data in batches of size 32 to improve computational efficiency and speed up convergence. Instead of all at once, the dataset is processed in batches, causing the model to update its weights more often, hence training faster.

Next, for each batch, the model's gradients are set to zero using `optimizer.zero_grad()`. Then comes a forward pass where `outputs = classifier(batch_x)` is used to calculate the model's predictions for the current batch. The loss is then calculated by comparing the predictions with the actual labels using `criterion(outputs, batch_y)`. When the loss is calculated, the backpropagation step is executed through `loss.backward()`, which computes the gradients of the loss with respect to the model's weights. Finally, the optimizer updates the model's weights using `optimizer.step()`, using the computed gradients to improve the model's performance.

Step 5: Evaluating the Model

First, it evaluates the model in eval mode to avoid updating weights. Then, it computes the predictions for each batch. After that, it converts the output to a discrete value using `torch.argmax`, storing it in a list. It then calculates accuracy and the classification report to measure the performance.

The model's predictions, `y_pred` are compared to the true labels, `y_test` in order to calculate the percent of correctly classified samples.

Print of the Loss for each epoch:

```
Epoch 1/10, Loss: 135.7253
..
Epoch 5/10, Loss: 67.7272
..
Epoch 10/10, Loss: 57.7541
```

These loss values show how much the model's predictions have in common with the real labels as the training goes on. The higher loss at the beginning—for example, the 135.7253 in Epoch 1—indicates that the model is making large mistakes, while the low loss at the end, for example, 57.7541 in Epoch 10, suggests that the model has

substantially improved its prediction quality during the course of training. A decreasing loss means the model is training itself, tuning its weights to reduce mistakes and increase its prediction skills.

```
Accuracy: 96.33%

Classification Report:
                   precision    recall   f1-score   support

  Legitimate          0.97      0.97      0.97          2493
    Phishing          0.96      0.95      0.95          1535

    accuracy                              0.96          4028
   macro avg          0.96      0.96      0.96          4028
weighted avg          0.96      0.96      0.96          4028
```

Comparison to Logistic Regression

The **Simple Neural Network (NN)** slightly outperforms the Logistic Regression model, achieving an accuracy of **96.33%** compared to Logistic Regression's **95.53%**.

For **legitimate messages**, the Simple NN achieves a **precision and recall of 0.97**, resulting in an **F1-score of 0.97**, matching its performance across the majority class.

For **phishing messages**, it achieves a **precision of 0.96**, a **recall of 0.95**, and an **F1-score of 0.95**, showing its ability to accurately detect phishing messages.

This small performance gain demonstrates that the Simple NN can learn slightly more complex patterns in the data compared to Logistic Regression while maintaining high precision, recall, and F1-scores across both classes.

Fully connected NN:

The Fully Connected Neural Network is more advanced Neural Network.The text data is converted into numeric features, and a PyTorch-based model is trained to make predictions. The FCNN contain 2 layers and one of them is hidden layer,
Also The FCNN uses activation function such as ReLU.
The downside of FCNN is that the computation cost of FCNN is higher than the SC.

Key differences between FCNN and SC:

Feature	SimpleClassifier	SimpleFCNN
Number of Layers	1	2
Hidden Layer	None	Yes (with configurable size)
Activation Functions	None (linear output only)	ReLU (hidden layer), Sigmoid (output)
Output	Raw logits for multi-class classification	Probabilities for binary classification
Complexity	Simple	More complex, can learn non-linear patterns
Use Case	Multi-class classification tasks	Binary classification tasks
Computation Cost	Lower	Higher

Step 1: TF-IDF Vectorization

The `TfidfVectorizer` is used to convert raw text into a numeric matrix, where each word is represented by its importance across documents. The maximum number of features is set to 150,000 to ensure efficient computation.

Step 2: Train-Test Split

The dataset is split into training (80%) and testing (20%) sets using `train_test_split`. The split ensures that the model can be evaluated on unseen data.

Step 3: Convert Data to PyTorch Tensors

The TF-IDF matrix and labels are converted into PyTorch tensors for compatibility with the neural network. The labels are reshaped to match the expected input dimensions for the loss function.

Step 4: Define the Fully Connected Neural Network

A simple feedforward neural network (`SimpleFCNN`) is defined using PyTorch:

- Layer 1: A fully connected layer (`fc1`) maps the input features to a hidden layer of size 128.
- ReLU Activation: Introduces non-linearity after `fc1`.
- Layer 2: Another fully connected layer (`fc2`) maps the hidden layer to a single output.
- Sigmoid Activation: Converts the output to probabilities for binary classification.

Step 5: Loss Function and Optimizer

- Loss: Binary Cross-Entropy Loss (`BCELoss`) is used to measure the difference between predicted probabilities and actual labels (0 for legitimate, 1 for phishing).
- Optimizer: The Adam optimizer adjusts the model's parameters to minimize the loss function, using a learning rate of 0.001 for stable convergence.

Step 6: Train the Model

- The model is trained over 10 epochs, with a batch size of 64 for efficient updates.
- At each step:
    - Data is shuffled for randomness.
    - The optimizer clears previous gradients (`optimizer.zero_grad()`).
    - Predictions are made with a forward pass through the network.
    - Loss is calculated, gradients are computed (`loss.backward()`), and the optimizer updates the weights.
- Training loss is printed at the end of each epoch to track progress.

Step 7: Evaluate the Model

The model is set to evaluation mode (`model.eval()`), and predictions are made on the test set. The predicted probabilities are converted to binary labels based on a

threshold of 0.5. Then, the accuracy and a detailed classification report are generated to evaluate performance.

```
Epoch [1/10], Loss: 0.1542
..
Epoch [5/10], Loss: 0.0022
..
Epoch [10/10], Loss: 0.0011

Accuracy: 0.9767

Classification Report:
              precision    recall  f1-score   support

  Legitimate       0.98      0.99      0.98      2493
    Phishing       0.98      0.96      0.97      1535

    accuracy                           0.98      4028
   macro avg       0.98      0.97      0.98      4028
weighted avg       0.98      0.98      0.98      4028
```

Comparison to Simple Neural Network and Logistic Regression

The **Fully Connected Neural Network** (**FCNN**) outperforms both the **Simple Neural Network** (**SNN**) and **Logistic Regression**, achieving an accuracy of 98% compared to 96.33% (SNN) and 95.53% (Logistic Regression).

For **legitimate messages**, the **FCNN** achieves a **precision** of 0.98, a **recall** of 0.99, and an **F1-score** of 0.98, indicating strong performance in identifying the majority class.

For **phishing messages**, it achieves a **precision** of 0.98, a **recall** of 0.96, and an **F1-score** of 0.97, demonstrating excellent detection of the minority class.

The results suggest that the added complexity of the FCNN, with its additional hidden layer, effectively enhances its ability to capture patterns in the data, making it a more robust model for phishing detection and other NLP tasks. This highlights the benefits of using slightly more complex architectures for such classification problems.

להוסיף על ה loss הקטן

Advanced models:

BeRT:

BERT (Bidirectional Encoder Representations from Transformers) is a transformer-based model that generates contextualized embeddings for text. Unlike traditional techniques like TF-IDF or Bag-of-Words, BERT captures the semantic meaning of words based on their context, making it highly effective for tasks like phishing detection.

How BERT Works

Step 1: Loading the Tokenizer and Model

The code will load the `AutoTokenizer` for BERT. This tokenizer is in charge of tokenizing raw text into words or subwords that the model can understand. This will make use of `bert-base-uncased`, which is a 12-layer version of BERT without any distinction between case (not sensitive to cases). Then, the code loads the BERT model itself (`AutoModel`), which is an untrained version of BERT that outputs embeddings (vector representations) for the given text.
BERT uses WordPiece tokenization for generating vector representations of each word in the text. It then uses Self-Attention to understand the relationship of different words in a bidirectional way, i.e., left to right and right to left.

Step 2: Text to Embedding

The function `text_to_Embedding` starts to converting the text into tokens using the tokenizer. The text is converted into a Tensor that can be fed into the model.
Embedding Calculation:

- The `last_hidden_state` from the model output is the representation of each token in the text after it has passed through the model.
- In particular, mean(dim=1) computes the average of all token embeddings, hence it gives one vector representation of the whole text.

The function `text_to_embedding` is applied to each text in the dataset to convert it into an embedding.
The embeddings are stored in the `embedded_data` list, which later will be converted to a PyTorch tensor.

Step 3: Defining the Classifier Model

A simple neural network model is defined with one fully connected (Dense) layer. This layer maps the embeddings to the number of output classes.
More details about this simple neural network model can be found in the previous part of the document.

Step 4: Loss and Optimizer Definition

The `CrossEntropyLoss` is used here, which is suitable for multi-class classification problems. The `AdamOptimizer` is used also to train the model effectively.

Step 5: Training Loop

The training loop divides the training data into batches and updates the model's weights by backpropagation. First, it conducts a forward pass over each batch in the model to obtain predictions. Then, it computes the loss by comparing these predictions with the actual labels. Thereafter, backpropagation is performed: it calculates gradients and updates the weights of the model such that the loss is minimal. This repeats itself many times—over many batches and epochs—to better tune the performance of the model.

Step 6: Evaluation of the Model and Classification Report

After training, the model's performance is evaluated on the test set. For each batch of test data, predictions are made and stored in the list y_pred.
Next, a classification report is printed, which shows the performance of the model on the test data, including metrics such as precision, recall, F1-score, and accuracy for each class

Classification Report:

	precision	recall	f1-score	support
Legitimate	0.97	0.97	0.97	2493
Phishing	0.95	0.95	0.95	1535
accuracy			0.96	4028
macro avg	0.96	0.96	0.96	4028
weighted avg	0.96	0.96	0.96	4028

Conclusion

The **BERT**-based model achieves high accuracy and balanced performance across both classes, showcasing its ability to handle complex textual data in phishing detection. However, its computational cost might outweigh its benefits in scenarios where simpler models (e.g., Logistic Regression with TF-IDF) already perform well. For tasks requiring deeper contextual understanding or handling highly imbalanced datasets, BERT is an excellent choice.

// להוסיף על ה Loss

BeRT + FCNN:

The **BERT + FCNN** approach leverages the power of **BERT** embeddings for contextual understanding, combined with a more advanced Fully Connected Neural Network (**FCNN**) to enhance classification performance. This setup introduces multiple hidden

layers with non-linear activation functions (**ReLU**), enabling the model to learn more complex patterns in the data.

How BERT + FCNN Works

1. **Embedding Extraction:**
    - **BERT** (`bert-base-uncased`) is used to generate embeddings for each email or SMS.
    - Tokenized text is passed through the BERT model to extract the mean of the last hidden state as the sentence-level embedding.
    - These embeddings serve as the **input features** for the FCNN.
2. **FCNN Architecture:**
    - The **FCNN** consists of:
        - A fully connected input layer mapping the BERT embeddings to a hidden layer of size 128.
        - A **ReLU** activation function applied to introduce non-linearity.
        - A second hidden layer with 64 neurons, followed by another ReLU activation.
        - An output layer mapping to the number of classes (legitimate or phishing).
    - Cross-Entropy Loss is used to measure performance, and the Adam optimizer is employed for efficient gradient updates.
3. **Training and Evaluation:**
    - The model is trained for 10 epochs with a batch size of 32.
    - Predictions are made on the test set, and evaluation metrics such as accuracy, precision, recall, and F1-score are calculated.

```
Classification Report:
              precision    recall  f1-score   support

  Legitimate       0.98      0.99      0.98      2493
    Phishing       0.98      0.97      0.97      1535

    accuracy                           0.98      4028
   macro avg       0.98      0.98      0.98      4028
weighted avg       0.98      0.98      0.98      4028
```

Strengths of BERT + FCNN:

- Enhanced Learning: The multiple hidden layers and ReLU activations in the FCNN allow the model to learn more complex relationships in the data.

- Balanced Performance: Achieves high precision, recall, and F1-scores for both phishing and legitimate messages, with an accuracy of 97%.
- Contextualized Embeddings: Leverages BERT's ability to capture the semantic meaning of text, improving classification accuracy.

Weaknesses of BERT + FCNN:

- Higher Computational Cost: The combination of BERT embeddings and a deeper FCNN increases the computational resources required for training and inference.
- Marginal Improvement: While it outperforms simpler models, the improvement is modest relative to the added complexity.

Final conclusion

This project demonstrates the effectiveness of various machine learning and deep learning techniques for phishing detection in emails and SMS messages. Below is a summary of the findings and conclusions:

1. Baseline Model (Dummy Classifier)

The Dummy Classifier serves as a baseline, achieving 61.3% accuracy, but it fails to detect any phishing messages (precision, recall, and F1-score of 0.00 for phishing). This highlights the need for more sophisticated models capable of identifying phishing messages effectively.

2. Traditional Models (Logistic Regression with TF-IDF)

- TF-IDF with Logistic Regression achieves an accuracy of 95.53%, with strong performance in both legitimate and phishing message classification. However, its recall for phishing messages (class 1) is slightly lower at 90%, indicating a few missed phishing detections.

3. Neural Networks (Simple NN and FCNN)

- Simple Neural Network achieves an accuracy of 96.33%, slightly outperforming Logistic Regression. It shows the ability to learn complex patterns from text features while maintaining balanced metrics across classes.
- Fully Connected Neural Network (FCNN) improves further, reaching 98% accuracy. Its additional hidden layer and ReLU activation functions enable it to learn more complex patterns, resulting in better detection of phishing messages (F1-score of 0.97 for phishing). However, the increased computation cost of FCNN needs to be considered.

4. Advanced Models (BERT and BERT + FCNN)

- BERT with Simple NN achieves an accuracy of 96%, showcasing the power of contextualized embeddings in capturing the semantic meaning of text. While effective, its computational cost is higher than traditional methods or simpler neural networks.

- BERT + FCNN achieves the highest accuracy at 98%, combining the contextual understanding of BERT with the learning capacity of FCNN. It demonstrates balanced performance across both classes, with an F1-score of 0.97 for phishing. Despite its computational demands, it is the most robust model tested in this project.

Model	Accuracy	Precision (Phishing)	Recall (Phishing)	F1-Score (Phishing)
Dummy Classifier	**61.3%**	**0.00**	**0.00**	**0.00**
Logistic Regression (TF-IDF)	**95.53%**	**0.98**	**0.90**	**0.94**
Simple Neural Network (TF-IDF)	**96.33%**	**0.96**	**0.95**	**0.95**
Fully Connected Neural Network (TF-IDF)	**98%**	**0.98**	**0.96**	**0.97**
BERT + Simple NN	**96%**	**0.95**	**0.95**	**0.95**
BERT + FCNN	**98%**	**0.98**	**0.97**	**0.97**